VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Optimalizace pomocí kolonií včel

Optimization Using Bee Colonies

2013

Jakub Šimůnek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Šimůnek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Optimalizace pomocí kolonií včel
Optimization Using Bee Colonies

Zásady pro vypracování:

Cílem práce je nastudovat a implementovat metody pro optimalizaci řešení založené na koloniích včel (ABC). Výsledkem bude jak základní optimalizační metoda tak i některé její varianty nalezené v literatuře. Schopnost hledat optimální řešení bude demonstrována na klasických optimalizačních problémech a na kompresi dat, kde bude použita pro hledání optimální abecedy pro kompresi textových dat.

Práce bude obsahovat:
1. Rešerši metod založených na ABC a jejich variantách, včetně jejich použití.
2. Návrh implementace.
3. Otestování výsledného algoritmu na klasických optimalizačních úlohách.
4. Otestování algoritmu v oblasti komprese dat.
5. Vyhodnocení výsledků experimentů.

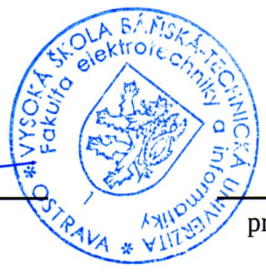
Seznam doporučené odborné literatury:

[1] R. Venkata Rao, Multi-objective optimization of multi-pass milling process parameters using artificial bee colony algorithm. Artificial Intelligence in Manufacturing, Nova Science Publishers, USA.
[2] R. S. Parpinelli, C. M. V. Benitez and H. S. Lopes; Parallel Approaches for the Artificial Bee Colony Algorithm; Handbook of Swarm Intelligence: Concepts, Principles and Applications. Series: Adaptation, Learning, and Optimization; Springer; pp: 329-346; 2011; Berlin, Germany.
[3] Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013 .....

Rád bych na tomto místě poděkoval Ing. Janu Platošovi, Ph.D., za pomoc a vedení při vypracování této práce.

Abstrakt

Diplomová práce se věnuje optimalizaci abecedy pro kompresi dat pomocí včelích kolonií. Nejprve práce obsahuje některé druhy včelích algoritmů a následnou implementaci. Obsahem je i testování na klasických optimalizačních funkcích. Práce také stručně popisuje základní typy kompresních algoritmů. V závěrečné části je proveden experiment spojující kompresi dat s včelími algoritmy.

Klíčová slova: včelí algoritmus; optimalizace; jedinec; fitness; komprese; kompresní poměr; kompresní algoritmus

Abstract

This diploma work is focused on the optimization of alphabet for data compression by using bee colonies algorithm. The first part of this work presents some types of bee algorithms and subsequent implementation. The content of this work is also testing on the classical optimization functions. The work concisely describes the basic types of the compression algorithms. The final section includes experiment which combines data compression with bee algorithm.

Keywords: bee algorithm; optimalization; individual; fitness; compression; compression ratio; data compression algorithm

Seznam použitých zkratek a symbolů

.NET	–	Platforma společnosti Microsoft
ABC	–	Artificial Bee Colony Algorithm
BA	–	Bees Algorithm
BCO	–	Bee Colony Optimization Algorithm
C#	–	Programovací jazyk

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Táto práce se věnuje optimalizaci abecedy pro kompresi textových dat. V našem moderním světě se s kompresí setkáváme téměř všude. Někteří lidé ani netuší že se komprese provádí přímo vedle nich. Mezi komprese, se kterými se setkal téměř každý patří například komprese zvuku(radio, televize) a obrázku(televize, fotky). I když to vypadá, že kompresní algoritmy jsou celkem na vrcholu, je potřeba otestovat některé nové optimalizační postupy.

Cílem této práce je optimalizovat abecedu(slovník) pro kompresi pomocí včelího roje. Optimalizace pomocí včelího roje patří do skupiny tzv. „hejnových algoritmů“, tyto algoritmy jsou popsány v kapitole 2.1. Nejprve se v práci zabývám různými typy včelích algoritmů, které by mohly být potencionálním řešení výše zmíněného problému. Tyto algoritmy jsou odvozeny z chování klasických včel medonosných. Mezi tyto algoritmy patří ABC, BA a BCO. Po objasnění základních včelích algoritmů následuje pohled na různé typy kompresních metod. Tyto typy jsou rozděleny na druhy podle funkčností. U každé kompresní metody je ukázán příklad komprese na krátkém řetězci. Dále se práce zabývá implementací včelích algoritmů a využitého kompresního algoritmu. V teto části popisuji dva způsoby implementace včelích algoritmů s popisem zjednodušených částí kódů. Popisuji zde i implementaci kompresního algoritmu, který jsem využil v programu. Následující kapitola se zabývá testováním algoritmů ABC a BA na klasických optimalizačních úkolech. Jsou zde popsány rozdíly těchto algoritmů a jejich chování. V poslední částí této práce je popis propojení včelího algoritmu s kompresním algoritmem. Jsou zde taktéž obsaženy výsledky komprese, které jsou porovnány s kompresním algoritmem LZW.

2 Včelí kolonie - Bee Colony

Algoritmus včelích kolonií patří do skupiny algoritmů označovaných jako „inteligence hejna“(swarm intelligence). Tato skupina je popsána v kapitole 2.1. Tento algoritmus je inspirován chováním skutečných včel při hledání potravy (Nektaru). Včely patří mezi tzv. sociální hmyz, který se vyznačuje tím, že mezi jedinci dochází k dělbě práce. Jinými slovy včely pracují jako tým, který dokáže řešit složité problémy. Jedinci mezi sebou komunikují pomocí tzv. „Včelího tance“. V této kapitole jsem využil poznatků ze zdrojů [5], [3].

2.1 Inteligence hejna

Mnohočlenná zvířecí společenství zvládají úlohy, na jaké by jedinec nikdy nestačil. V posledních letech se velice rozmohlo využití tzv. „inteligence hejna“. Tyto algoritmy jsou založeny na kolektivním chováním různých společenství (hejn, kolonií, atd). Vývoj algoritmů spočíval v pozorování chování zvířecích společenství v určitých situacích. Většina současných algoritmů je realizována pro optimalizační úkoly. Mezi nejznámější z optimalizačních hejnových algoritmů patří tyto:

1. Optimalizace hejnem částic
2. Optimalizace mravenčí kolonií
3. Optimalizace včelím rojem
4. Optimalizace hejnem světlušek

Hejnové algoritmy jsou schopny pracovat v n-dimenzionálním prostoru a vyznačují se tím, že jsou schopny najít výsledek blížící se požadovanému, za krátkou dobu.

2.2 Obecné chování včel

Při přestěhování kolonie do nového prostředí je třeba nalézt zdroje potravy. Této části se ujímají včelí průzkumníci, kteří se rozlétnou po okolí a hledají potravu. Jakmile průzkumník nalezne potravu sesbírá z nalezené oblasti nektar. Poté se vrací do úlu, kde sesbíraný nektar odevzdá dělnicím. V této části se průzkumník rozhoduje co dál dělat. Má na výběr 3 možnosti:

- Průzkumník poletí opět na náhodné místo (Pokud bylo na nalezeném místě málo nektaru).
- Průzkumník bude hledat v okolí nalezeného místa, zda tam nejsou nějaké větší zdroje nektaru.
- Průzkumník letí zpět na nalezené místo a přitom se snaží předat informaci kolegyním. Tuto informaci dává pomocí tzv. „Včelího tance“ na speciálním místě v úlu.

Při včelím tanci se včela „tanečnice“ snaží přilákat větší počet vyčkávacích včel. Ostatní včely sledují tanečnice. Včela sledující tanec se může přidat k některé z tanečnic a letět do její oblasti, nebo se sama vydá na průzkum.

2.3 Umělé včely

Umělé včely jsou chováním velmi podobné včelám živým. Na počátku jsou všechny včely v úlu. Následně se vyšlou průzkumníci do prostoru, kde hledají informaci(například hodnotu funkce). Rozdíl nastává při předávání informací mezi průzkumníkem a vyčkávací včelou. Při předávání informací se nepoužívá včelí tanec. To znamená, že se komunikuje přímo. Existuje větší množství algoritmů. Některé zde popíši.

2.3.1 Artificial Bee Colony Algorithm (ABC)

Artificial Bee Colony Algorithm lze přeložit jako algoritmus umělého včelího roje. Využívá se zde tří druhů včel: dělnice, průzkumníci a vyčkávací včely. Na začátku algoritmu průzkumníci vyletí z úlu do prostoru, kde získají nějaké prozatímní řešení, a stávají se z nich dělnice. Dělnice pracují na svém prozatímním řešení a snaží se ho vylepšit pomocí lokálního prohledávání. Dále se dělnice snaží přilákat vyčkávací včely. Tyto vyčkávací včely si s větší pravděpodobností vybírají dělnici s lepším řešením. Přiřazené vyčkávací včely se pomocí lokálního vyhledávaní snaží vylepšit řešení jejich dělnice. Pokud nějaká vyčkávací včela nalezne lepší řešení než má aktuální dělnice,tak si dělnice své řešení aktualizuje. V případě, že dělnice nezlepší své řešení po několik iterací, zahodí své řešení a stane se průzkumníkem. Zahodit své řešení může dělnice pouze pokud není její řešení v množině nejlepších řešení. Tato množina obsahuje určité procento dělnic s nejlepšími řešeními. Bez této množiny by mohla včela, která dosáhla nejlepšího možného řešení, po určitém počtu iteracích zahodit své řešení(protože už by lepší řešení neexistovalo). Pro úspěšné hledáni je potřeba nastavit několik parametrů:

- Počet průzkumníků.
- Velikost kolonie.
- Celkový počet iterací.
- Počet iterací, kdy včela má opustit své řešení, jestliže ho nezlepšila.
- Rozsah, ve kterém má algoritmus počítat.
- Další parametry podle použití. Například hranice ve kterých se má počítat, velikost lokálního prohledávání atd.

Nyní popíši algoritmus pomocí pseudokódu.

```
X: Řešení aktuální dělnice.
Y: Nejlepší řešení z přiřazených včel
Čítač − čítač jak dlouho včela nezlepšila řešení
Vygenerovaní populace(Rozeslaní průzkumníku po prostoru).
for i 1 < Iterace
   begin
      Seřazení dělnic podle aktuálního řešení.
      cyklus − postupné procházení dělnic.
         begin
            if (Čítač < konstanta)
              Přiřazení počtu vyčkávacích včel dělnici.
               Rozeslaní přiřazených včel po okolí dělnice.
               if (Y lepší než X)
                  Nahrazení aktuálního řešení dělnice lepším řešením
                  Čítač na aktuální dělnici nastaven na 0
               else
                  Čítač na aktuální dělnici zvýšen o 1
                  Vráceni aktuálního jedince.
            else
               Změna dělnice na průzkumníka a vyslání na náhodnou pozici v prostoru
         end
   end
```

Výpis 1: ABC

2.3.2 Bees Algorithm (BA)

Bees Algorithm lze přeložit jako včelí algoritmus. V algoritmu se využívá dvou druhů včel: průzkumníků a vyčkávacích včel. Průzkumníci se na začátku rozmístí do prostoru, kde získají prozatímní řešení. V další části algoritmu se průzkumníci rozdělí do dvou skupin podle jejich částečných řešení. První skupina je skupina s lepšími řešeními a druha skupina obsahuje horší řešení. Vyčkávací včely se přiřazují pouze mezi včely v lepší skupině, a to podle kvality jejich částečného řešení. V další čísti algoritmu se včely z lepší skupiny snaží za pomoci přidělených vyčkávacích včel zlepšit své aktuální řešení. Včely z horší skupiny své řešení zahodí a opět se stávají průzkumníky, kteří jsou opět vysláni na náhodnou pozici. I u tohoto algoritmu je třeba nastavit některé parametry.

- Počet průzkumníků.
- Velikost kolonie.
- Celkový počet iterací.
- Počet včel s lepším řešením(Tuto část můžeme nastavit i poměrem např. půl na půl)
- Rozsah, ve kterém se má počítat.
- Další parametry podle použití. Například hranice, ve kterých se má počítat, velikost lokálního prohledávání atd.

Algoritmus se v základu podobá algoritmu ABC popsaného v kapitole 2.3.1 V následujícím pseudokódu lze vidět podobnost algoritmů.

```
X: Řešení aktuální dělnice.
Y: Nejlepší řešení z přiřazených včel
Z: Počet včel s lepším řešením
Vygenerovaní populace(Rozeslání průzkumníku po prostoru).
for i 1 < Iterace
   begin
      Seřazení dělnic podle aktuálního řešení.
      cyklus − postupné procházení dělnic.
         begin
            if ( i < Z)
               Přiřazení počtu vyčkávacích včel průzkumníkovi.
               Rozeslání přiřazených včel po okolí dělnice.
               if (Y lepší než X)
                  Nahrazení aktuálního řešení dělnice lepším řešením
               else
                  Vráceni aktuálního řešení.
            else
               Zahozeni řešení a vyslání včely na náhodnou pozici.
         end
   end
```

Výpis 2: BA

2.3.3 Bee Colony Optimization Algorithm (BCO)

Název lze přeložit jako optimalizační algoritmus včelím rojem. Roj není rozdělen do žádných skupin. Algoritmus obsahuje dvě fáze.

1. Dopředná fáze.
2. Zpětná fáze.

Při dopředné fázi se snaží včely vylepšovat své částečné řešení. Ve zpětné fázi se včely vrací do úlu, kde porovnávají své částečné řešení s ostatními včelami. Včely s neuspokojivým částečným řešením své řešení zahodí a převezmou řešení některé více úspěšné včely. Včely s velmi dobrým částečným řešením dostávají větší pravděpodobnost, že se k ní přidá některá včela se zahozeným řešením. Všechny tyto fáze se provádějí v iteracích, dokud není splněna ukončovací podmínka. Jednoduchý popis algoritmu BCO je zobrazen na následujícím pseudokódu.

```
B − Počet včel v úlu.
NC − počet kroků při dopředné fázi.
Na počátku algoritmu jsou všechny včely v úlu.
Všechny včely se nastaví na prázdné řešení.
```

```
for i 1 < Iterace
    begin
        Dopředná fáze:
            Nastavení čítače k = 1.
            cyklus − dokud není splněná podmínka if $(k > NC)$.
                Zhodnocení všech možných kroků.
                Náhodně vybrat další posun.
                k = k + 1   // navýšení čítače o 1.
        Zpětná fáze:   // Všechny včely se vrátí do úlu.
            Seřazení včel podle částečného řešení.
            Rozhodování zda včela bude pokračovat ve svém řešení, nebo své řešeni zahodí.
            Přiřazeni řešení včelám bez řešení.
    end
Výpis nejlepšího řešení.
```

Výpis 3: BCO

Tento algoritmus můžeme využít na větší množství problémů. Například pro hledání maxima (minima) funkce, nebo na problém obchodního cestujícího.

3 Komprese

Komprese dat(také Komprimace dat) je proces, při kterém redukujeme velikost bloku informací. Při kompresi se snažíme snižovat redundanci dat. S různými typy kompresí se můžeme setkat téměř všude(televize, rádio, fotky, atd.). První známé komprese dat vznikly daleko dříve, než se začaly používat počítače. Jednou z nejvíce známých kompresí je Morseova abeceda. Více o Morseově abecedě v sekci 3.2

Kompresi lze rozdělit na dva hlavní druhy.

Druhy kompresí:

- **Bezeztrátová komprese** - je způsob komprese, kdy nedochází ke ztrátě informací. Pokud tedy data dekomprimujeme, tak budou v původním stavu.
- **Ztrátová komprese** - je způsob komprese, při kterém některá data, z důvodu zlepšení kompresního poměru, zahodíme nebo pozměníme. Při dekompresi se tedy nikdy nedostaneme do původního stavu. Jinak řečeno, budou chybět nějaká data. Tato komprese se nejčastěji využívá u obrázků a videí a zvuku. Při kompresi se využívá fyziologických omezení lidských smyslů. Kompresní metody lze taktéž rozdělit do několika skupin.
  Typy kompresních metod:
  - Statistické metody.
  - Slovníkové metody.
  - Komprese obrazu, videí a zvuků.

3.1 Obecná komprese

3.2 Morseova abeceda

Morseovu abecedu vymyslel na konci 19. století americký fyzik Samuel F. B. Morse (1791–1872), který ji v roce 1844 vyzkoušel při prvním telegrafickém spojení. V tabulce 1 je zobrazena Morseova abeceda. Znaky mají délku kódu závislou na četnosti používání v anglických textech. Například E je v anglických textech nejpoužívanější znak. Proto má E nejkratší kód. Bez komprese by měl každý znak stejně dlouhý kód a posílaní by bylo velmi pomalé. Tím, že mají znaky délku kódu závislou na četnosti výskytů v anglických textech, urychlují zasílání zpráv, jelikož nejčetnější znaky mají krátký kód.

3.3 RLE(run length encoding)

RLE je jedna z nejjednodušších bezeztrátových kompresních metod. Funkce spočívá v nahrazení po sobě se opakujících znaků dvojicí počet a znak. Například řetězec *aaaabbbccccaa* zakódujeme do následujícího textu *4a3b4c2a*. Na tento algoritmus existují různé modifikace. Například pokud budeme mít řetězec *abbcc*, základní algoritmus by ho prodloužil na *1a2b2c*. Proto existuje úprava, kde znaky, které nedokážeme zmenšit necháme jak jsou. Ovšem při této úpravě je potřeba nějak označit, zda se jedná o zakódovaný, nebo o

Znak	Kód	Znak	Kód	Znak	Kód
A	.-	I	..	R	.-.
B	-...	J	.- - -	S	...
C	-.-.	K	-.-	T	-
D	-..	L	.-..	U	..-
E	.	M	- -	V	...-
F	..-.	N	-.	W	.- -
G	- -.	O	- - -	X	-..-
H	....	P	.–.	Y	-.- -
CH	- - - -	Q	- -.-	Z	- -..

Tabulka 1: Morseova abeceda

nezakódovaný znak. Toto rozhodování se provádí přidáním znaku, který přidáme před zakódované části.Tento znak má v sobě uloženou informací, že následuje kódovaná fráze.

RLE je vhodné na data, která obsahují dlouhé sekvence stejných znaků. Proto se nejčastěji používá jako pomocná metoda u komprese obrazu a videí. Nejvhodnější z dat jsou kreslené obrázky, které obsahují dlouhé sekvence stejné barvy.

3.4 Statistické metody komprese dat

Statistické metody jsou metody, které využívají pravděpodobnost výskytů znaků k jejich zakódování.

3.4.1 Entropie

Jelikož data jsou navržená tak, aby se s nimi dalo lehce pracovat, obsahují velké množství redundancí. Metody pro ukládaní dat většinou neberou ohled na části, které se opakují. Proto potřebujeme daleko větší prostor pro uložení dat. Například program, který využívá ASCII tabulku, s hodnotami 0-127, ukládá data po osmi bitech, zatímco pro uložení znaku stačí bitů 7. Nejjednodušší komprese může tedy odstranit nepoužívaný bit, čímž snížíme velikost dat o $\frac{1}{8}$.

Všeobecně je komprese metoda, kdy se snažíme redundance odstraňovat a tímto snížíme i velikost dat.*Z pohledu komprese nás tedy zajímá, zda délka dat je odpovídající vzhledem k množství informace obsažené v datech*[2]. Míru informace můžeme zjistit pomocí entropie dat. Na následujícím vzorci 1 je zobrazena průměrná entropie v bitech.

Předpoklady:

symboly - $a_1, a_2, ..., a_n$

Pravděpodobnost výskytu symbolu - $P_1, P_2, ..., P_n$

Výpočet průměrné entropie:

$$E = -\sum_{i=1}^{n} P_i \log_2 P_i \tag{1}$$

Entropie je tedy míra informací v prvku dat. Při kódování nám pomáhá zjistit do jaké míry je kód optimální. Pro příklad si vytvoříme slovo obsahující 4 znaky. Slovo *abcdabcd* obsahuje tyto znaky $A = (a, b, c, d)$ s pravděpodobnostmi výskytu $P = (0.25, 0.25, 0.25, 0.25)$. Entropie je tedy $E = -4(0.25 \log_2 0.25) = 2$. Slovo tedy můžeme zakódovat pomocí dvoubitových kódů (00,01,10,11) a kód bude optimální. Ovšem pokud vezmeme slovo *aabbcdaa*, které obsahuje stejné znaky jako předchozí slovo, jeho pravděpodobnosti výskytu jsou jiné $P = (0.5, 0.25, 0.125, 0.125)$. Entropie takového slova je $E = -0.5 \log_2 0.5 - 0.25 \log_2 0.25 - 2(0.125 \log_2 0.125) = 1,75$. Kdybychom použili dvoubitového kódu jako v předešlém případě, nebyl by kód optimální. Pro takovéto případy je potřeba využít kódy proměnlivé délky. Kódové značení znaků může vypadat např. takto (1,01,000,001). Kódování je zapotřebí provést tak, aby jsme byli schopni bezchybně dekomprimovat data.

3.4.2 Huffmanovo kódování

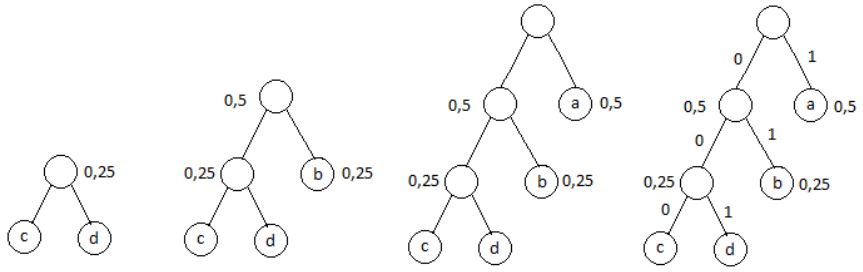
Huffmanovo kódování patří k nejstarších a nejznámějších kódováním. Bylo vyvinuto Davidem Huffmanem v roce 1952. Princip je založen na četnosti znaků. Znakům,které mají největší četnost, se přidělují nejkratší kódy. Kódovaní lze provádět více způsoby. Nejjednodušší způsob je pomocí binárního stromu.
Konstrukce Huffmanova kódu:
Předpokládejme, že text se skládá z n znaků $a_1, a_2, a_3, ..., a_n$, a nechť tyto znaky jsou seřazeny tak, že pravděpodobnosti jejich výskytů $p_1, p_2, p_3, ..., p_n$ tvoří nerostoucí posloupnost (tj. $p_i \geq p_{i+1} proi = 1, 2, ..., n-1$)[2]. Jinými slovy seřadíme všechny znaky, obsažené v datech, podle jejich četností.

- Nejprve vezmeme dva znaky s nejmenší pravděpodobností a vytvoříme z nich strom. Každý list bude pojmenován daným znakem. Kořen bude pojmenován součtem pravděpodobností těchto dvou znaků.
- Následně do posloupnosti pravděpodobností přidáme nově vytvořený kořen. Znaky, které tento kořen z posloupnosti vymažeme, jelikož už jsou zakódovány.
- Opět vybereme dva znaky (nebo stromy) s nejmenší pravděpodobností a vytvoříme uzel se součtem jejich pravděpodobností. Pokud je více možností, zvolíme náhodně jednu z nich. Pokračujeme předchozím krokem, dokud seznam pravděpodobností nebude prázdný.
- Směrem od kořene k listům ohodnotíme potomky 0 a 1.

Znak	Četnost	Pravděpodobnost	Kód
a	4	0,5	1
b	2	0,25	01
c	1	0,125	000
d	1	0,125	001

Tabulka 2: Tabulka znaků s četnostmi a kódy

Praktická ukázka:
Mějme slovo *aabbcdaa*. Toto slovo obsahuje znaky (a,b,c,d) s četnostmi (4,2,1,1) a tedy s pravděpodobnostmi (0.5,0.25,0.125,0.125). V tabulce 2 jsou zobrazeny znaky s četnostmi a jejich kódování. Kódování je vytvořeno pomocí Huffmanova stromu zobrazeného na obrázku 1.



Obrázek 1: Postupné sestaveni Huffmanova stromu

Nejprve se spojí dva znaky s nejmenší pravděpodobností. Vznikne tedy podstrom s hodnotou součtu pravděpodobností, která je v příkladu 0,25. Následně spojíme následující dvě pravděpodobnosti, což je náš podstrom s hodnotou 0,25 a znak b s hodnotou 0,25. Nakonec do stromu stejným způsobem přidáme i znak a. Jako poslední krok ohodnotíme potomky bizarními čísly 0 a 1. Nyní pomocí stromu můžeme zakódovat znaky. Kódy znaků jsou zobrazeny v tabulce 2.

Nyní můžeme zkusit vypočítat průměrnou entropii slova *aabcdaa*. Po dosazení do vzorce vyjde entropie 1,75 bitů. Tedy že pro optimální kód je potřeba 1,75 bitů na zapsání znaku. Pokud dané slovo zakódujeme pomocí tabulky 2 vznikne nám posloupnost o délce 14 bitů 11010000010111. 14 bitů pro zapsání osmi znaků znamená, že na zapsání jednoho znaku potřebujeme 1,75 bitů. Takže naše kódování je optimální.

Výhodou Haufmanova kódování je rychlost komprese i dekomprese. Nevýhodou je nutnost uložení binárního stromu, aby jsme byli schopni data dekomprimovat. Díky této nevýhodě se může toto kódování projevit až u větších dat.

3.4.3 Aritmetické kódování

V aritmetickém kódování používáme k zakódování celého textu jedno číslo z intervalu $< 0, 1)$ Základní předpoklady jsou stejné jako u Huffmanova kódování. Pro začátek je

Znak	Četnost	Pravděpodobnost
a	4	0,5
b	2	0,25
c	1	0,125
d	1	0,125

Tabulka 3: Tabulka znaků s pravděpodobnostmi

Krok	Načtený symbol	Interval od	Interval do
0	-	0	1
1	a	0	0,5
2	a	0	0,25
3	b	0,125	0,1875
4	c	0,171875	0,1796875
5	a	0,171875	0,17578125
...	...	...	...

Tabulka 4: Postupné tvoření intervalů

potřeba mít vypsané všechny znaky a jejich pravděpodobnosti výskytu. Aritmetické kódování následně rozdělí interval $< 0, 1)$ na množinu disjunktních intervalů. Počet těchto disjunktních intervalů je roven počtu znaků. Velikost intervalů odpovídá pravděpodobnosti výskytu znaků.

Postup kódování

- Nejdříve si sepíšeme četnosti znaků a vypočítáme jejich pravděpodobnost.
- Nastavíme interval na $< 0, 1 >$.
- Přečteme znak a interval se rozdělí na podintervaly dle pravděpodobnosti znaků.
- Interval nastavíme na hodnotu, kterou má disjunktní interval u přečteného znaku. Dále pokračujeme předchozím bodem, dokud nebude zakódovaný celý text.
- Jakmile je zakódován celý text, vybereme číslo z konečného intervalu a toto číslo zapíšeme.

Praktická ukázka:
Použijeme slovo *aabcadab*, ke kterému máme pravděpodobnosti znaků zapsané v tabulce 3. Podle těchto pravděpodobností můžeme rozdělovat intervaly. Výsledek kódování je náhodné číslo z konečného intervalu. Toto číslo zapíšeme.

Dekódování probíhal podobným způsobem jako při kódování. Taktéž využíváme intervalů. Máme přečtenou hodnotu a pomocí této hodnoty určujeme v každém kroku, do kterého intervalu tato hodnota patří.

	$< 0, 1)$	$< 0, 0.5)$	$< 0, 0.25)$	$< 0.125, 0.1875)$
a	$< 0, 0.5)$	$< 0, 0.25)$	$< 0, 0.125)$	$< 0.125, 0.15625)$
b	$< 0.5, 0.75)$	$<, 0.25, 0.375)$	$< 0.125, 0.1875)$	$< 0.15625, 0.171875)$
c	$< 0.75, 0.875)$	$< 0.375, 0.4375)$	$< 0.1875, 0.0.21875)$	$< 0.171875, 0.1796875)$
d	$< 0.875, 1)$	$< 0.4375, 0.5)$	$< 0.1875, 0.0.25)$	$< 0.1796875, 0.1875)$

Tabulka 5: Intervaly pro hledání slova

Postup dekódování:

- Nastavíme interval na $< 0, 1 >$.
- Rozdělíme aktuální interval dle pravděpodobností.
- Vybereme interval do kterého naše hledané číslo patří a zapíšeme znak patřící k danému intervalu. Tento interval si zapíšeme a pokračujeme předchozím krokem dokud nebude dekomprimován veškerý obsah.

Praktická ukázka dekomprese:
Pro příklad jsem vybral číslo 0.172 z intervalu $< 0.171875, 0.17578125 >$, který jsme zjistili při kompresi. S tímto čísle jsme schopni dojít k výslednému slovu *aabc*. Pomocí tabulky 5 tedy dokážeme vypsat řetězec do původního (bezeztrátového) tvaru. Začínáme tedy v prvním sloupci. Naše hodnota 0.172 patří do intervalu ke znaku *a*. Tento znak si napíšeme. Interval do kterého patří naše číslo vypíšeme do sloupce 2. Rozdělíme interval a najdeme disjunktní interval, do kterého patří naše číslo. Číslo patří do intervalu u znaku *a*. Písmeno *a* tedy připíšeme a vzniká nám slovo *aa*. Interval posuneme do dalšího sloupce, kde ho znovu rozdělíme dle pravděpodobnosti. Tímto způsobem pokračujeme, dokud nenalezneme celý řetězec.

Aritmetické kódování je pomalejší než Huffmanovo kódování. Ovšem dokáže si lépe poradit s velmi nerovnoměrným rozložením četností a dosahuje lepších výsledků. U aritmetického kódování se každým znakem velikost intervalu zmenšuje. Tím pádem naše desetinné číslo, přepsané do bitové podoby, je s každým znakem delší, což zhoršuje práci s těmito čísly. Proto byly vyvinuty metody aritmetického kódování s celými čísly, nebo taky s omezením na počet desetinných míst.

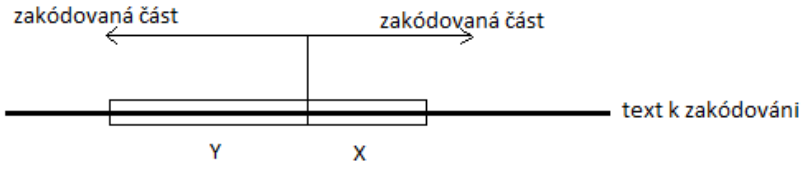
3.5 Slovníkové metody

Slovníkové metody hledají opakující se fráze v textech. Slovníkové metody se rozdělují do dvou skupin:

1. První skupina jsou metody, kdy si vytváříme slovník a fráze v textu nahrazujeme indexem dané fráze obsažené ve slovníku.
2. Další skupina jsou metody, kdy fráze nahrazujeme odkazem na místo, kde se tato fráze poprvé vyskytla.

3.5.1 LZ77

Metoda LZ77 hledá opakující se fráze do určité délky. Na obrázku 2 je zobrazen postup metody LZ77.

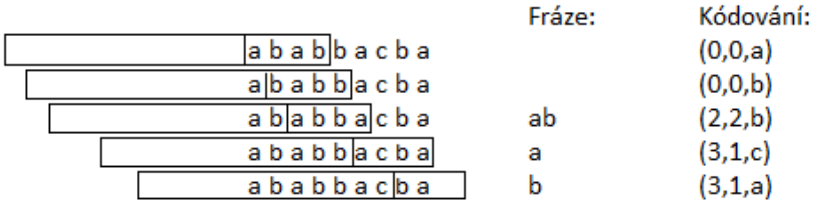


Obrázek 2: Náhled na posuvné okénko

Posuvné okénko(sliding window) je rozděleno na dvě části(X a Y). V části X je doposud nekódovaný text. Část Y obsahuje již zakódovaný text. Toto okénko se posouvá po textu, který chceme kódovat. Postup kódování je prováděn následovně. V části Y vyhledáváme fráze odpovídající frázím za části X. Jestliže je nalezen větší počet frázi, vybereme tu nejdelší. Pokud je nalezeno více nejdelších frází, vybereme tu, která je nejblíže od mezníku mezi X a Y. Frázi zakódujeme následujícím způsobem. (i,j,znak), kde *i* je vzdálenost od mezníku mezi X a Y, *j* je délka fráze a *znak* je znak následující za frází. Po zakódování fráze posuneme okénko o *j+1* znaků doprava. Při kódovaní můžou nastat 2 mezní případy

- Nenalezneme žádnou frázi. Zapíšeme tedy (0,0,znak) a posuneme okénko o 1 doprava.
- Nalezneme frázi shodnou se znaky v celé části X. V tomto případě zapíšeme frázi bez posledního znaku.

Na obrázku 3 je zobrazen příklad s postupem kódování pro jednoduchý text. Dekomprese



Obrázek 3: Postup kódování LZ77

z kódu je velice jednoduchá. Zapisujeme si pouze dekomprimovaný text. V tomto textu vyhledáváme fráze pomocí kódů. Dekomprese je daleko rychlejší, jelikož nemusíme vyhledávat fráze v textu. Máme přesnou pozici i délku. Příklad na dekompresi je zobrazen na obrázku 4.

Kódování:	Dekódovaný text:
(0,0,a)	a
(0,0,b)	ab
(2,2,b)	ababb
(3,1,c)	ababbac
(3,1,a)	ababbacba

Obrázek 4: Postup dekomprimováni LZ77

3.5.1.1 Metoda LZSS Tato metoda byla vytvořena v roce 1982 Jamesem Storerem and Thomasem Szymanskim. LZSS vznikla úpravou algoritmu LZ77. Na rozdíl od předcházející metody vychází z myšlenky, že odkaz fráze může být delší než řetězec, který chceme nahradit. Dále LZSS neukládá za nalezenou frází následující znak, jelikož by tímto znakem mohla začínat další fráze. Pokud se nenalezne fráze větší než jeden znak, ukládáme samostatný znak. Abychom při dekompresi věděli jestli čteme zakódovanou frázi, nebo pouze zakódovaný znak, musíme ukládat ještě jeden bit, který bude označovat zda se jedna o frázi, nebo znak. Zápis tedy má 2 části:

- (rozeznávací bit, i,j) - uložení odkazu na frázi
- (rozeznávací bit, znak) - pokud není nalezena fráze

Rozeznávací bit nabývá hodnot 0 a 1. Je to bit, pomocí kterého poznáme při dekódování, jestli se jedná o frázi, nebo o znak. Následující znak i je vzdálenost od prvního nezakódovaného znaku a j je délka fráze.

3.5.2 LZ78

LZ78 je metoda publikována v roce 1978 Abrahamem Lempelem a Jacobem Zivem. Princip je jednoduchý. Všechny fráze si ukládáme do slovníku a přiřazujeme jim postupně čísla od 1. Text na vstupu porovnáváme s frázemi ve slovníku. Vybereme nejdelší frázi a text zakódujeme následujícím způsobem:

- (číslo_fráze,znak), kde *číslo_fráze* je číslo nejdelší nalezené fráze a *znak* je nasledujíc znak.
- (0, aktuální_znak), kde *aktuální_znak* je první znak, který je právě na vstupu. Tento zápis se používá, pokud není nalezena žádná fráze. Jednoduše řečeno zakódujeme samostatný znak.

Po každém zakódování dvojice přidáme do slovníku další frázi, která odpovídá nalezené frázi + následujícímu znaku. Tato fráze je tedy o jeden znak delší než původní.

Při každém kroku je do slovníku přidána další fráze. Jelikož je velikost slovníku omezená, musí se slovník po dosažení plného stavu vymazat. Po vymazání začínáme následující text kódovat znovu s prázdným slovníkem. Na obrázku 5 je zobrazen postup kódování pomocí LZ78.

		Nová fráze	
Řetězec	Kód	Číslo fráze	Fráze
ababbacbac	(0,a)	1	a
babbacbac	(0,b)	2	b
abbacbac	(1,b)	3	ab
bacbac	(2,a)	4	ba
cbac	(0,c)	5	c
bac	(4,c)	6	bac

Obrázek 5: Postup komprese pomocí LZ78

3.5.2.1 LZW Metoda LZW je vylepšení metod LZ78. Toto vylepšení bylo publikováno v roce 1984 Abrahamem Lempelem, Jacobem Zivem a Terry Welchem. Stejně jako u LZSS (viz. podkapitola 3.5.1.1) je hlavní myšlenkou nekódovat znak za frází, jelikož by tento znak mohl být využitelný v následující frázi. LZW kóduje pouze fráze délky 2 a větší. Proto musíme do slovníku před začátkem přidat všechny fráze délky 1. Jinak řečeno musíme do slovníku přidat všechny znaky, které jsou použity v textu. U číslování frází nemusíme začínat číslem 1(jako u LZ78), ale můžeme začít od 0. Nepotřebujeme totiž identifikátor, který nám ukáže, že jsme nenašli žádnou frázi.

Pří každém kroku, mimo první krok, přidáme do slovníku další frázi. Tato fráze se skládá z fráze z minulého kroku a počátečního písmena aktuální fráze. Postup kódováni slova *ababbacbac* je zobrazen na obrázku 6 Výstupem této metody je posloupnost 0,1,4,5,2,7.

Dekomprese se provádí obdobným způsobem jako komprese. Z dekomprimovaného textu si postupně skládáme identický slovník se slovníkem použitým při kompresi. Dekomprese je zobrazena na obrázku 7. První fáze je obdobná jako u komprimace zobrazené na obrázku 6. Jak lze vidět, opět jsme si při každém kroku(mimo prvního) přidávali do slovníku další frázi.

3.6 Algoritmus využitý v této práci

Pro tuto práci jsem využil kompresní algoritmus, který se skládá z částí různých algoritmů. Hlavní myšlenkou je vytvořit slovník obsahující fráze o délce 1, 2, ...,n, kde n je uživatelem definovaná maximální délka. Jinak řečeno můžu si zvolit, jestli budu používat znaky, dvojznaky, nebo delší fráze. Ze začátku musí slovník obsahovat všechny použité znaky, aby jsme byli schopni zakódovat celý text. Slovník má omezenou délku.

Fáze 1 - přidání všech znaků do slovníku

Index	Fráze
0	a
1	b
2	c

Fáze2 - Kódování textu

	Přidání fráze		Kódování	
Text	Číslo	Fráze	Fráze	Výstup
ababbacbac	-	-	a	0
babbacbac	4	ab	b	1
abbacbac	5	ba	ab	4
bacbac	6	abb	ba	5
cbac	7	bac	c	2
bac	8	cb	bac	7

Obrázek 6: Postup komprese pomocí LZW

Fáze2

	Přidání fáze		
Vstup	Číslo	Fráze	Výstup
0	-	-	a
1	4	ab	b
4	5	ba	ab
5	6	abb	ba
2	7	bac	c
7	8	cb	bac

Obrázek 7: Postup dekomprese pomocí LZW

Jelikož musíme slovník ukládat, nesmí byt moc velký. Jakmile máme slovník vytvořený, postupně procházíme řetězec a nacházíme nejdelší možnou frázi. Tuto frázi nahrazujeme pozicí fráze ve slovníku. Slovník se tvoří jednoduše. Nejprve projdeme data a přidáme do slovníku všechny fráze délky 1. Následně procházíme text znovu, ale po dvojznacích. Takto pokračujeme podle nastavení délky frází. Na obrázku 8 lze vidět postup vytvoření

Řetězec :ababbacbac

Slovník:

Kód	Fráze	Kód	Fráze
0	a	5	cb
1	b	6	ac
2	c	7	aba
3	ab	8	bba
4	ba	9	cba

Komprese:

Řetězec:	Fráze	Výstup
ababbacbac	aba	7
bbacbac	bba	8
cbac	cba	9
c	c	2

Obrázek 8: Postup komprese pomocí algoritmu užitého v práci

slovníku a komprese řetězce. Výstup komprese je posloupnost indexů 7, 8, 9, 2. Tedy z původních deseti znaků jsme počet znaků snížili na čtyři. Jelikož slovník používá indexy 0-9, nepotřebujeme k ukládání výstupu 8 bitů na jeden znak(index). Postačí nám pouze 4 bity. Musíme ovšem také počítat s uložení slovníku. Optimalizace slovníku je asi největší slabina tohoto algoritmu. Pokud se podíváme na obrázek 8, lze vidět, že při komprimaci nebyly použity žádné dvojznaky. Tedy pokud by ve slovníku tyto dvojznaky nebyly, ušetřilo by nám to dost místa. O optimalizaci slovníku se pokouším v kapitole 6.

Jelikož si můžeme načíst uložený slovník, dekomprese je velice jednoduchá. Načteme si tedy slovník a postupně procházíme posloupnost čísel. Tato čísla jsou indexy ve slovníku. Indexy nahradíme frází a dekomprese je hotová.

4 Implementace

Pro implementaci jsem využil programovací jazyk C #.

4.1 Jazyk C# a jeho vlastnosti

C# je jednoduchý, objektově orientovaný programovací jazyk vytvořený společností Microsoft. Tento jazyk je plně kompatibilní s technologií .NET a je odvozen od jazyků C a Java. Nevýhodou jazyka C# je programování časově náročné aplikace, protože jazyk nemá některé klíčové komponenty pro aplikace s velmi vysokým výkonem. Vlastnosti Jazyka C#:

- plná podpora tříd a objektově orientovaného programování, včetně dědičnosti rozhraní i implementace, virtuálních funkcí a přetížení operátorů,
- konzistentní a vhodně definovaná sada základních typů,
- integrovaná podoba automatického generování dokumentace ve formátu XML,
- automatické čištění dynamicky přidělování paměti
- možnost označit třídy nebo metody uživatelsky definovanými atributy. Tato funkce může být užitečná pro dokumentaci a někdy má určitý vliv na překlad (např. při označení metod, aby se překládaly pouze v ladicích sestaveních),
- plný přístup ke knihovně základních tříd .NET a také snadná dostupnost rozhraní Windows API (pokud ho skutečně potřebujete, k čemuž nedochází příliš často),
- v případě potřeby jsou dostupné ukazatele a přímý přístup do paměti, ale jazyk je navržen takovým způsobem, že lze bez nich pracovat téměř ve všech situacích,
- podpora vlastností a událostí ve stylu jazyka Visual Basic,
- pouhou změnou možností překladače můžete překládat buď spustitelný soubor, nebo knihovnu komponent .NET, kterou může volat jiný kód stejným způsobem jako ovládací prvky ActiveX (komponenty COM)
- pomocí jazyka C# lze psát dynamické stránky ASP.NET a webové služby založené na XML [6]

4.2 Využité komponenty

V této části popíši některé komponenty, které jsem využil

4.2.1 Zedgraph

Zedgraph [1] je volně dostupná knihovna tříd pro vytváření 2D čárových, sloupcových a koláčových grafů. Knihovna je stavěna jak pro ASP, tak i pro Windows Form. Dále je zedgraph velmi flexibilní. V grafu můžeme upravovat takřka vše. Ovšem pro jednoduché použití má většinu možností nastavenou na nějakou defaultní hodnotu.

Nastavením grafu se zabývají 2 hlavní prvky. *MasterPane*, který slučuje všechny grafy, a *GraphPane*, pomocí kterého nastavujeme konkrétní grafy. Především název grafu, názvy os, typy os, minima a maxima os, nastavení hlavních i vedlejších mřížek a mnoho dalšího. Důležité metody pro nastavení grafu:

- *AxisChange( )*- Nastavuje rozsahy os tak, aby byly všechny vykreslené body v tomto rozsahu.
- *RestoreScale( )* - Resetuje nastavení os na původní.
- *ZedGraph.Refresh()* - Aktualizuje daný graf.
- *GraphPane.AddCurve()* - Přidá do grafu křivku.
- *ZoomOutAll()* - Zruší veškerá přiblížení a nastaví pohled na původní.
- *GraphPane.Title.Text*- Název grafu.
- *GraphPane.XAxis.Title.Text*-Název osy X.
- *GraphPane.YAxis.Title.Text*- Název osy Y.

4.2.2 Mersenne Twister

Mersenne Twister[4] je volně dostupný generátor náhodných čísel, který vivinuli Makoto Matsumoto a Takuji Nishimura v roce 1996/1997. Dále pak vylepšen v roce 2002.
Výhody generátoru:

- Navržení s ohledem na nedostatky ostatních generátorů
- Daleko větší posloupnost čísel
- Rychlá generace
- Dobré využití paměti

Při testování včelích algoritmů pomocí tohoto generátoru byly výsledky optimalizačních úloh daleko přesnější. Pomocí klasického generátoru jsem se mnohdy ke správnému výsledku nedostal.

	Jedinec1	Jedinec2	Jedinec3	Jedinec4	Jedinec5	...
Fitness	0,5256456	0,237798	0.48641	0.84351	0.2635	
Čítač	0	0	0	0	0	
Parametr1	2.15476	45.48413	6.787765	25.57645	65.48314	
Parametr2	1.784165	0.14564	65.94589	0.57485	25.6453	
Parametr3	10.54646	5.85314	10.21456	2.547685	2.41556	
Parametr4	85.4636	1.85765	4.47896	2.84223	0.54314	
...						

Tabulka 6: Populace pomocí dvourozměrného pole

4.3 Implementace ABC

Implementace tohoto algoritmu není moc složitá. Pokud budu postupovat podle pseudokódu 1, je nejdříve zapotřebí vytvořit počáteční populaci. Populace se dá naprogramovat více způsoby. První způsob je pomocí dvourozměrného pole. Teto způsob je zobrazen v tabulce 6. Každý sloupec označuje jednoho jedince. Tento jedinec obsahuje na nultém řádku fitness(hodnota funkce v daném bodě). Řádek s indexem 1 obsahuje čítač. V tomto políčku je zapsán počet iterací od posledního zlepšení. Ostatní řádky obsahují parametry. Jsou to náhodné hodnoty v rozmezí definičního oboru funkce. Jinak řečeno se jedná souřadnice bodu. Z těchto souřadnic se počítá fitness. Tento způsob zápisu jedinců se mi zdá nepřehledný a také složitý. Při implementaci jsem využil objektů. Vytvořil jsem si třídu Individual(jedinec). Tato třída obsahuje proměnnou *fitness*, proměnnou *count*(čítač) a $List< double > dim$, který slouží pro uložení souřadnic, ze kterých se počítá fitness. Tato třída je zobrazena na výpisu kódu 4.

```
public class Individual
{
    public double fitness;
    public int count;
    public List<double> dim = new List<double>();
}
```

Výpis 4: Třída individual

Tento objekt ukládám do listu objektů(List<Individual>), což tvoří mnou vygenerovanou populaci. S takto vygenerovanou populací lze jednodušeji pracovat a také se dá jednodušeji upravovat. Například přidání nějaké proměnné jedinci daleko jednodušeji než u řešení pomocí pole, kde by jsme museli přepsat část programu.

Populaci máme vytvořenou a algoritmus pokračuje cyklem, který se opakuje dokud není dosažen určitý počet iterací. Na začátku každé iterace se musí včely seřadit podle fitness hodnoty. U jedinců zapsaných v poli je potřeba napsat vlastní třídící algoritmus, což není složité. Při používání objektů můžeme taktéž vytvořit vlastní třídící algoritmus, nebo můžeme využít funkci „Sort()“. Pomocí této funkce lze setřídit jedince velmi jednoduše. Setřízení jedinců pomocí metody „Sort()“ je zobrazeno na výpisu kódu 5.

```
  oldPop.Sort(delegate(Individual ind1, Individual ind2)
{
  return ind1.fitness.CompareTo(ind2.fitness);
});
```

Výpis 5: Setřízení podle hodnoty fitness

Následuje cyklus procházející populací. Ke každé dělnici je potřeba přiřadit určitý počet následovníků. Toto rozdělení lze provést více způsoby. Můj způsob je velice jednoduchý. Včely rozděluji podle jednoduchého vzorce.

Vzorec pro přiřazení včel:

$$vcely = ((d - j) * (\frac{v}{\sum_{i=0}^{d} i})) + 1 \tag{2}$$

Kde d znamená počet dělnic, v je počet vyčkávacích včel a j je index aktuálního jedince. Na konci vzorce přičítám 1, jelikož počítám s tím, že okolí bude prohledávat i samotná dělnice. Pro lepší pochopení vypočítám přiřazení včel na konkrétním příkladu.

- 10 průzkumníků(později dělnic)
- 400 vyčkávacích včel
- j=5

$$vcely = ((10 - 5) * (\frac{400}{\sum_{i=0}^{10} i})) + 1 = 37 \tag{3}$$

To znamená, že k dělnici s indexem 5 přiřadíme 37 vyčkávacích včel. Pro správnou funkci tohoto vzorce je potřeba dbát na správné nastavení vstupních parametrů. Při špatném nastavení poměru mezi dělnicemi a vyčkávacími včelami by mohla nastat situace, kde vyčkávací včely budou přiřazeny pouze k několika nejlepším dělnicím.

Jelikož vyčkávací včely máme rozděleny, můžeme započít s lokálním prohledáváním. Nejprve ověříme jestliže včela není v lokálním extrému. Toto ověření lze provést pomocí počítadla, jež má každá dělnice. Hodnota tohoto čítače se inkrementuje vždy, když včela nezlepší své aktuální řešení. Jestliže je hodnota čítače větší, než nastavená mez, dělnice své řešení zahodí a odletí na náhodnou pozici. Pro správnou funkci je potřeba zajistit, aby pár nejlepších včel nezahazovalo řešení. Bez tohoto ošetření by mohla nastat situace, při které by dělnice, která je již v globálním extrému, mohla zahodit své řešení. Pokud jsou tyto podmínky splněny, tak může začít lokální prohledávání. Na Výpisu kódu 6 je zobrazen postup lokálního prohledávání. Toto prohledávání začíná cyklem, kde proměnná *bees* značí počet přiřazených vyčkávacích včel.

```
for (int i = 0; i < bees; i++)
{
    do
    {
        stav = true;
        Activ = new Individual();
        for (int k = 0; k < dimension; k++)
        {
            Activ.dim.Add(oldPop[j].dim[k] - range + (ranGen.NextDouble() *
            ((oldPop[j].dim[k] + range) - (oldPop[j].dim[k] - range))));

            // ověření, zda nový parametr patří do definičního oboru funkce
            if ((Activ.dim[k] > highbound) || (Activ.dim[k] < lowbound))
            {
                stav = false;
            }
        }
        if (stav) break;
    } while (true);
    // Zde patří výpočet fitness z nových
    ActualInd.Add(Activ);
}
```

Výpis 6: Lokální prohledávání

Následující cyklus pouze ověřuje zda-li nově vytvořené parametry náleží definičnímu oboru funkce. Poslední cyklus vytváří nové hodnoty parametrů. Tyto hodnoty jsou z blízkosti aktuální dělnice. Nový parametr vzniká jednoduchým způsobem. Nejdříve je potřeba si načíst parametr aktuální dělnice. Pomocí tohoto parametru vytvoříme hranice. Dolní hodnota hranice se vytvoří odečtením nastavené hodnoty pro velikost lokálního prohledávaní od parametru aktuální dělnice. U horní hranice je to totéž, ale s přičtením nastavené hodnoty. Jednoduchý příklad určení hranic je zobrazen v tabulce 7. Když už jsou vypočteny hranice, následuje vygenerování náhodného čísla z námi vytvořeného intervalu. Tyto výpočty se provedou pro každý parametr. V poslední části je potřeba vypočítat fitness z nových parametrů a následné uložení do prozatímního listu jedinců. Tyto výpočty provedeme pro všechny přiřazené včely. Pro konečnou fázi je potřeba najít nově vytvořenou včelu s nejlepší fitness hodnotou. Tuto fitness hodnotu porovnáme s fitness hodnotou aktuální dělnice. Pokud je fitness aktuální dělnice lepší, inkrementujeme čítač dělnice o 1 a ponecháme dělnici se svým řešením. Pokud fitness aktuální dělnice není lepší, nahradíme řešení dělnice řešením nejlepší vyčkávací včely a čítač nastavíme na 0.

Po provedení lokálního prohledávání u všech dělnic začíná nová iterace a celý proces se opakuje. Ukončení algoritmu může nastat po určitém počtu iterací, nebo při splnění ukončovací podmínky.

	Dělnice	Dolní hodnota	Horní hodnota
Fitness	0,5256456		
Čítač	0		
Parametr1	7	2	13
Parametr2	10	5	15
Parametr3	15	10	20
Parametr4	11	6	16

Tabulka 7: Lokální prohledávání

4.4 Implementace BA

Pokud porovnáte pseudokód 2 s pseudkódem 1 tak zjistíte velikou podobnost. Začátek implementace algoritmu, až po přidělení včel, je obdobný s algoritmem ABC popsaným v kapitole 4.3. U BA algoritmu se provádí rozdělení pouze pro včely s lepšími výsledky. V mém případě používám jako počet lepších včel první polovinu průzkumníků. Vzorec pro přidělení vyčkávacích včel je tedy podobný vzorci 2.

$$vcely = ((p - j) * (\frac{v}{\sum_{i=0}^{p/2} i})) + 1 \tag{4}$$

Kde p znamená počet průzkumníků, v je počet vyčkávacích včel, j je index aktuálního jedince a $p/2$ je součet indexů první poloviny průzkumníků včel.

Následuje lokální prohledávání. Toto prohledávání okolí se provádí pouze pro včely z lepší skupiny. Včely z horší skupiny své aktuální řešení zahodí a vygenerují si novou náhodnou pozici. Na včelách z lepší skupiny se tedy provádí lokální prohledávání. Kód lokálního prohledávání u BA je obdobný s lokálním prohledáváním u ABC. Tento kód je zobrazen je výpisu kódu 6.

4.5 Implementace binary ABC

Základní algoritmus ABC popsaný v kapitole 4.3 není uzpůsoben pro práci s binárními čísly. pro správnou funkčnost je potřeba algoritmus lehce upravit. Nejprve je potřeba vytvořit základní populaci. V této náhodné populaci používáme jako parametry hodnoty 0 nebo 1. Nejjednodušší postup jak vygenerovat populaci je pomocí náhodného čísla v intervalu $< 0, 1 >$. Toto náhodné číslo porovnáme s konstantou 0,5. Pokud je $nhodnslo < 0,5$ parametr se nastaví na 0. V opačném případě, když $nhodnslo \geq 0,5$, nastavíme parametr na 1. Kód je zobrazen ve výpisu 7.

```
  for (int j = 0; j < dimension; j++)
{
    if ((ranGen.NextDouble()) < 0.5)
    {
        ind.dim.Add(0);
        if (find.dim[j] == 1)
        {
            M01++;
        }
    }
    else
    {
        ind.dim.Add(1);
        if (find.dim[j] == 0)
        {
            M10++;
        }
        else
        {
            M11++;
        }
    }
}
ind.fitness = 1 - (double)((double)M11 / (double)(M10 + M01 + M11));
```

Výpis 7: Vytváření náhodného jedince

Následuje výpočet fitness hodnoty. Pro testování jsem použil jako fitness hodnotu vzdálenost jedince od určitého vektoru. Tuto vzdálenost lze počítat více způsoby. V mém programu používám vzorec pro výpočet Jaccardova indexu. Mějme tedy vektor $X = (x_1, x_2, x_3, ..., x_D)$ a vektor $Y = (y_1, y_2, y_3, ..., y_D)$, kde D je dimenze vektoru. Při porovnání vektorů můžou nastat 4 situace:

- $x_d = y_d = 1$
- $x_d = 1, y_d = 0$
- $x_d = 0, y_d = 1$
- $x_d = y_d = 0$

Kde *d* nabývá hodnot 1, 2, ..., D. tyto situace si pojmenujeme následně:

- M_{11} je počet všech bitů, kde bit v X a zároveň Y je roven 1($x_d = y_d = 1$)
- M_{10} je počet všech bitů, kde bit v X je 1 a bit v Y je 0 ($x_d = 1, y_d = 0$)
- M_{01} je počet všech bitů, kde bit v X je 0 a bit v Y je 1 ($x_d = 0, y_d = 1$)
- M_{00} je počet všech bitů, kde bit v X a zároveň Y je roven 0($x_d = y_d = 0$)

Nyní můžeme tyto hodnoty dosadit do Jaccardova vzorce:

$$fitness = 1 - \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \tag{5}$$

Čím menší fitness je, tím podobnější jsou si vektory X a Y. Při fitness = 0 platí, že vektory X a Y jsou si rovny.

Základní populace je již vytvořena. Populaci setřídíme podle fitness hodnot. Přiřazení počtu vyčkávacích včel je obdobné jako u ABC popřípadě BA. Lokální prohledávání je od předešlých algoritmů jiné. Základní princip zůstává stejný. Podle počtu následovníků vygenerujeme určitý počet nových včel, ze kterých vybereme včelu s nejlepší fitness hodnotou a tuto včelu porovnáme s aktuálním řešením. Nové včely se vytvářejí pomocí následujícího vzorce:

$$X^j = D_i^j \oplus [\alpha(D_i^j \oplus N_k^j)] \tag{6}$$

Kde X^j je j-ta dimenze nově vytvořené včely, D_i^j je j-ta dimenze i-te dělnice, N_k^j je j-ta dimenze k-te včely(Je náhodně vybrána včela z množiny dělnic, kde i $\neq$ k) a α je šance na negaci výrazu. Šanci na negaci(α) jsem nastavil mezi $(0, 2 >$ procenta. Nejednodušší řešení, jak této pravděpodobnosti dosáhnout, je vygenerováním náhodného čísla v intervalu $< 0, 1 >$ a následném porovnání s konstantou např 0,2. Při tomto nastavení algoritmus dosahoval nejlepších výsledků. Při testování jsem nezkoušel pouze XOR ($\oplus$), ale i AND a OR. Operátor XOR v mém případě dával nejlepší výsledky. Abych urychlil algoritmus pro vytváření nových okolních včel, vytvářím nové včely ve vláknech. Jelikož se včely vytvářejí nezávisle, můžeme použít vlákna. Vytvoření vláken je zobrazeno na výpisu kódu 8

```
 resetEvents = new ManualResetEvent[bees];
for (int i = 0; i < bees; i++)
{
    ThreadInfo t = new ThreadInfo();
    t.i = i;
    t.j = j;
    resetEvents[i] = new ManualResetEvent(false);
    ThreadPool.QueueUserWorkItem(new WaitCallback(MakeNew), t);
}

foreach (var item in resetEvents)
{
    item.WaitOne();
}
```

Výpis 8: Prohledávání okolí

Nejdříve je potřeba vytvořit objekt *Threadinfo*, do kterého si uložíme jako je index aktuální dělnice a číslo přiřazené včely. Dále potřebujeme pole *resetEvents*, ve kterém se můžeme dozvědět, zda-li je vlákno hotové. Následně vytvoříme *threadpool*, ve kterém voláme metodu *MakeNew*. Tato metoda vytváří novou včelu. Před pokračováním algoritmu je potřeba počkat, než se všechna vlákna provedou. Na tuto kontrolu slouží cyklus *foreach*

Počet vyčkávacích včel	doba bez vláken $[ms]$	doba s vlákny $[ms]$
200	2212	1232
500	4992	2057
800	7521	2849
1100	10360	3712
1400	12957	4585
1700	15437	5422
2000	18300	6300

Tabulka 8: Časové porovnání algoritmu s vlákny a algoritmu bez vláken při změně počtu vyčkávacích včel

zobrazený v kódu. V tabulce 8 je zobrazený časový rozdíl mezi algoritmem s vlákny a algoritmem bez vláken. V této tabulce je zobrazena pouze závislost času na změně vyčkávacích včel. Tyto časy jsou počítány při hodnotách:

- Průzkumníku 20
- Iterace 300
- dimenze 300
- vyčkávací včely je proměnná

Jak lze vidět z tabulky 8, při zvýšení počtu vyčkávacích včel se jsou vlákna čím dál více užitečnější. Obdobný případ nastává i při změně dimenze. Časy algoritmu při změně dimenze jsou zobrazeny v tabulce 9. Testování těchto časů proběhlo při parametrech:

- Průzkumníku 20
- Iterace 300
- dimenze je proměnná
- 1000

Na posledním řádku tabulky 9 lze vidět zpoždění algoritmu bez vláken přibližně o 20s. Jestliže si spojíme obě tabulky (9, 8) dospějeme k tomu, že vlákna opravdu velmi zrychlují algoritmus.

V každém vláknu se spouští metoda *MakeNew*, ve které se vypočítává nová včela z okolí. Tato včela se počítá pomocí vzorečku 6. Po dokončení všech vláken přichází na řadu vybrání nejlepšího jedince. Jedinec se vybírá z listu nově vytvořených jedinců. Nejlepšího jedince porovnáme s aktuálním jedincem(dělnicí). Jestliže je aktuální dělnice lepší, přičteme k jejímu čítači 1. Pokud je dělnice horší, nahradíme prozatímní řešení řešením z nejlepšího nového jedince.

Celý tento algoritmus opakujeme po určitý počet iterací, nebo dokud není splněna ukončovací podmínka.

Dimenze	doba bez vláken $[ms]$	doba s vlákny $[ms]$
100	232	196
200	7032	2634
300	9681	3511
400	12485	4434
500	15349	5326
600	18002	6399
700	21286	7471
800	23948	8379
900	26751	9322
1000	29747	10297

Tabulka 9: Časové porovnání algoritmu s vlákny a algoritmu bez vláken při změně dimenze

4.6 Implementace kompresního algoritmu

Tato část se zabývá implementací kompresního algoritmu popsaného v kapitole 3.6. Začátek je velice jednoduchý. Pomocí streamu si otevřu daný soubor. Tento soubor si po znacích načtu do paměti.Následně můžu začít vytvářet slovník. Tento slovník je inicializován tabulkou a jeho velikost je omezena(zadaná velikost). Vzhled je zobrazen v tabulce 10. Každý řádek této tabulky je jedna fráze. Sloupce tabulky jsou znaky v dané frázi.

```
for (int i = 0; i < data.Count − pocetznaku; i++)
{
    for (int j = rows − 1; j >= 0; j−−)
    {
        stav = true;
        for (int k = 0; k < pocetznaku; k++)
        {
            delka = k;
            if (slovnik[j, 0] == −1) { stav = false; break; }
            if (slovnik[j, k] == −1) { delka = k − 1; break; }
            if ((data[i + k] != slovnik[j, k]))
            {
                stav = false;
                break;
            }
        }
        if (stav)
        {
            i += (delka);
            dataout.Add(j);
            break;
        }
    }
}
```

Výpis 9: Vyhledávání nejdelších frází ve slovníku

První znak	Druhý znak	Třetí znak	...
a	-1	-1	
b	-1	-1	
c	-1	-1	
a	b	-1	
c	a	-1	
a	b	c	
atd.			

Tabulka 10: Tvorba slovníku

Slovník:		Uložení slovníku:	
Kód	Fráze	Krok	Uložený text
0	a	1	a
1	b	2	ab
2	c	3	abc
3	ba	4	abcba
4	aba	5	abcbaaba
5	bba	6	abcbaababba

Tabulka 11: Ukládání slovníku

Může nastat situace, že slovník zcela nezaplníme. Například nastavíme využívání fráze délky 1. V takovém případě zmenšíme slovník tak, aby nezůstaly volné řádky. Slovník tedy máme hotový a můžeme začít s kompresí. V následujícím kódu jsou zobrazeny cykly pro vyhledávání nejdelších frázi ve slovníku.

První cyklus zobrazený ve výpisu 9 prochází znaky, které jsou načtené v paměti(v Listu). Následující cyklus prochází slovník od konce. Tedy od nejdelších frází. Poslední cyklus porovnává znaky z dat se znaky ve slovníku a snaží se najít nejdelší shodu.

Data jsou přepsána na posloupnost indexů, takže můžeme přejít k zápisu. Než zapíšeme slovník a data je potřeba zapsat některé informace. Na začátek souboru tedy zapíšeme počet řádků slovníku a počet frází délky 1. Také počty frází délky 2 a počty delších frází. Jakmile máme tyto hodnoty zapsány, můžeme začít ukládat slovník. Postupně tedy ukládáme znaky po bytech. Ukládání je zobrazeno v tabulce 11. Ukládáme tedy fráze po znacích postupně za sebou. Jelikož jsme si zapsali počty frází s určitými délkami, načtení slovníku při dekompresi bude snadné. Jakmile uložíme celý slovník, přichází na řadu ukládání dat(posloupnosti indexů). Než začneme ukládat data, je potřeba zjistit kolik bitů potřebujeme k zapsání jednoho znaku(indexu). Tuto informaci si můžeme zjistit pomocí slovníku. Zjistíme tedy index poslední fráze ve slovníku. Tímto známe největší číslo, které potřebujeme zapsat. Vypočteme si tedy kolik bitů potřebujeme k zapsání tohoto čísla. Počet bitů pro ukládání tedy známe a můžeme začít ukládat data. Při dekompresi dokážeme načíst slovník, takže si dokážeme vypočítat po kolika bitech máme data číst.

5 Testování ABC a BA

V této kapitole testuji algoritmy ABC a BA na klasických optimalizačních funkcích. V první částí popisuji detailněji rozdíl algoritmů na první de Jongově funkci. V Následné testuji algoritmy na dalších funkcích. Jako testovací funkce jsem použil optimalizační funkce z knihy [3]

5.1 Podrobnější testování

Zde následuje podrobnější testování algoritmů ABC a BA na jedné z optimalizačních funkcí.

5.1.1 Testování ABC(DeJong 1)

Algoritmus jsem odzkoušel na několika vícerozměrných funkcích(De Jong, Rastrigin's function, Schwefel's function a další). Pro podrobnější popis jsem si vybral první de Jongovu funkci popsánou v kapitole 5.1.3, Pro výpočet globálního minima v první de Jongově funkci jsem parametry nastavil na následující hodnoty:

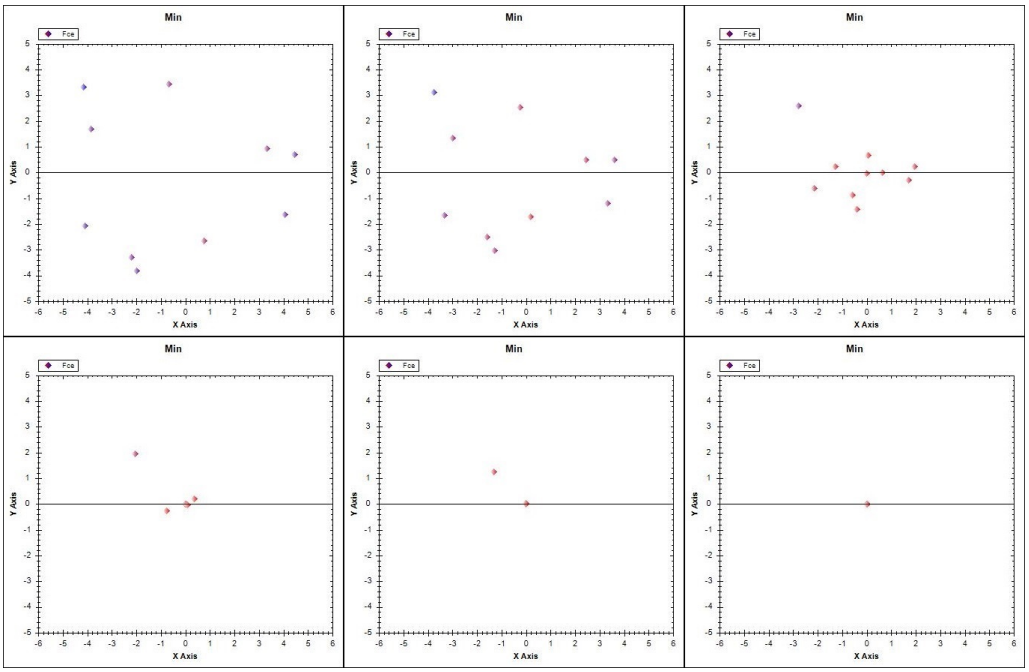
- Počet průzkumníku =10
- Velikost kolonie = 120
- Počet iterací = 200
- Prohledávané okolí =0,1
- Počet iterací, kdy včela má opustit své řešení = 10
- Počítání v intervalu $< -5, 5 >$
- Dimenze = 2

Na obrázku 9 je zobrazen průběh hledání globálního minima. Na tomto obrázku lze vidět, že algoritmus bez problému našel globální minimum.

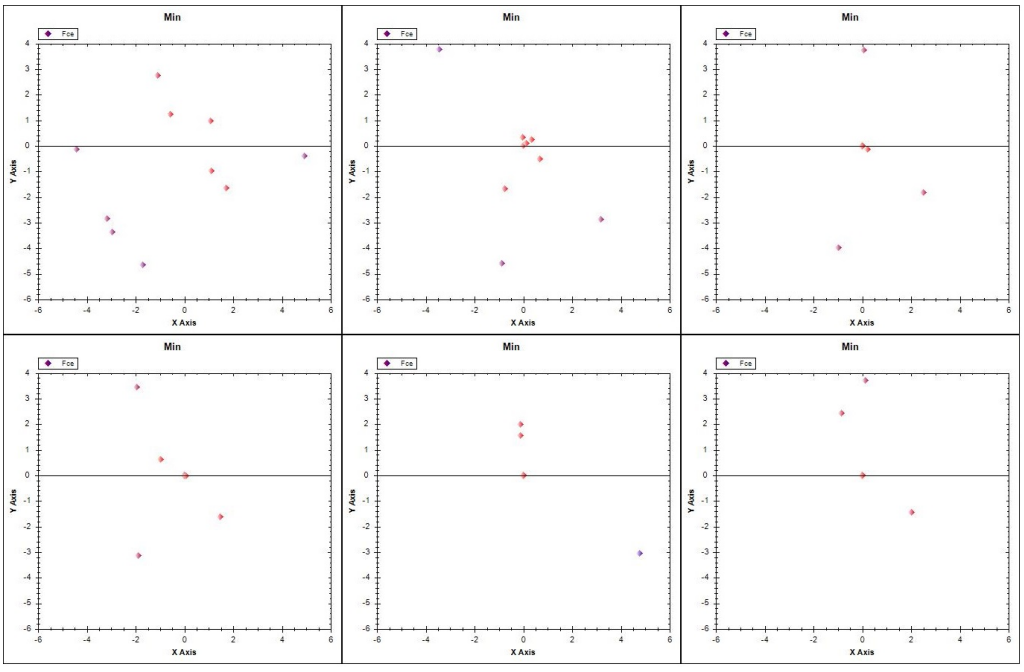
5.1.2 Testování BA(DeJong 1)

Tento typ algoritmu jsem testoval na obdobných funkcích jako algoritmus ABC(viz kapitola 5.1.1). Pro porovnání s předchozím algoritmem jsem pro ukázku opět použil první de Jongovou funkci popsanou v kapitole 5.1.3. Nastavení parametru je taktéž obdobné jako při testování ABC.

- Počet průzkumníku =10
- Velikost kolonie = 120
- Počet iterací = 200



Obrázek 9: ABC:Průběh pohybu včel při iteracích (0, 10, 30, 50, 70, 110)



Obrázek 10: BA:Průběh pohybu včel při iteracích (0, 10, 30, 50, 70, 110

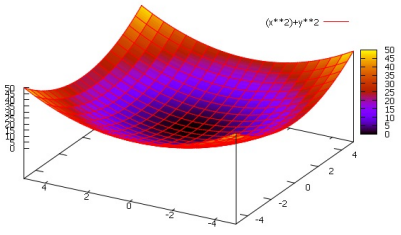
- Prohledávané okolí =0,1
- Počet včel v lepší skupině = Počet průzkumníků/2
- Počítání v intervalu $< -5, 5 >$
- Dimenze = 2

Na obrázku 10 je zobrazen průběh hledání globálního minima první de Jongovy funkce. Lze vidět, že algoritmus minimum našel velice rychle. Už při iteraci č. 30 se některé včely blížili globálnímu minimu. Při porovnání obrázku 10 s obrázkem 9 můžeme vidět, jak algoritmus BA se k minimu dostal daleko dříve. Toto dřívější dosažení minima je způsobeno přidělením většího počtu vyčkávacích včel, včelám s lepším řešením. Zatímco u ABC se vyčkávací včely rozdělují mezi všechny průzkumníky, tak u BA se vyčkávací včely rozděluji jenom mezi včely s lepším řešením.

5.1.3 První de Jongova funkce

Tato funkce je první ze čtyř de Jongových funkcí. Je popsáná následujícím vzorcem:

$$\sum_{i=1}^{D} x_i^2 \tag{7}$$



Obrázek 11: Zobrazení první de Jongovy funkce

Globální minimum první de Jongovy funkce, zobrazené na obrázku 11 se nachází v bodě $(x_1, x_2, ..., x_n) = (0, 0, ..., 0)$.

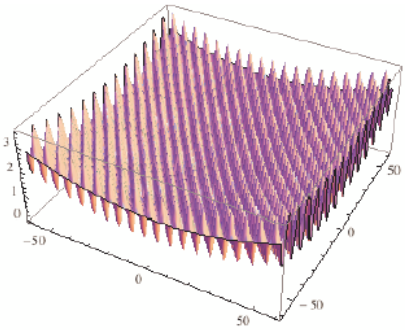
5.2 Testování algoritmů na dalších funkcích

V této podkapitole je ukázáno chování algoritmů BA a ABC na dalších optimalizačních funkcích. Výsledky testování jsou v této podkapitole stručnější. Především jsem se zaměřil na kvalitu vyhledání minima.

5.2.1 Griewangkova funkce

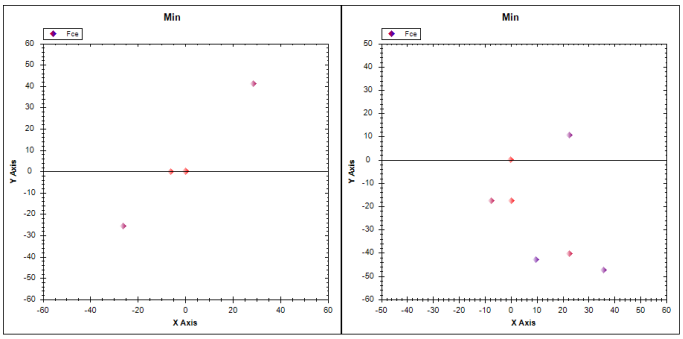
$$1 + \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos \frac{x_i}{\sqrt{i}} \tag{8}$$

Griewangova funkce je popsaná vzorcem 8. Tato funkce je zobrazena na obrázku12 a



Obrázek 12: Zobrazení Griewangkovy funkce

její globální minimum se nachází v bodě $(x_1, x_2, ..., x_D) = (0, 0, ..., 0)$, kde D je dimenze. Hodnota globálního minima je 0 pro jakoukoli dimenzi. Na obrázku 13 je zobrazená

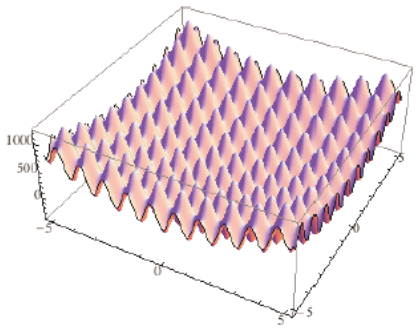


Obrázek 13: Griewang:Rozvržení jedinců při poslední iteraci. ABC(vlevo), BA(vpravo)

poslední iterace obou algoritmů při jednom z několika testů. Po provedení několika testů jsem vypozoroval, že oba algoritmy se velice přibližují hledanému globálnímu minimu. U algoritmu ABC se k minimu přibližuje větší počet včel, ale s menší přesností. Zatímco u algoritmu BA se přibližuje menší počet včel s větší přesnosti. Jako příklad bych uvedl hodnoty nejlepších včel. Včela ABC=0,0016, včela BA=0,0003. Důvod lepšího výsledku BA je nejspíše dán rychlejším přiblížením ke globálnímu minimu. Toto je způsobeno větším počtem vyčkávacích včel u nejlepšího jedince.

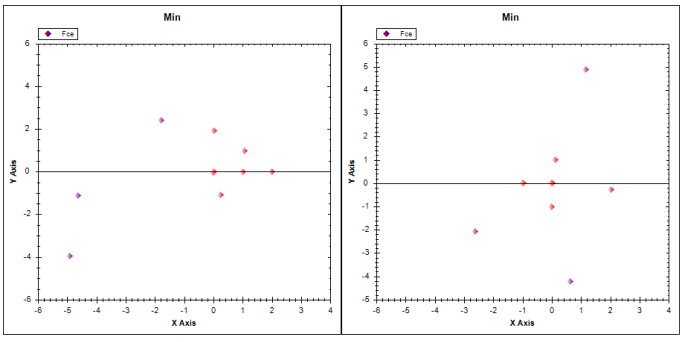
5.2.2 Rastriginova funkce

$$10D \sum_{i=1}^{D} x_i^2 - 10 \cos 2\pi x_i \tag{9}$$



Obrázek 14: Zobrazení Rastriginovy funkce

Rastriginova funkce popsána vzorcem 9 je zobrazena na obrázku 14. Pro D=2 je globální minimum v bodě $(x_1, x_2) = (0, 0)$ a hodnota tohoto minima je -400. Pro větší dimenzi je globální minimum v bodě $(x_1, x_2, ..., x_D) = (0, 0, ..., 0)$, kde D je dimenze. Hodnota minima pro větší dimenzi je $y = -200 \times D$. I když výsledky hledání minima(zobrazeny na



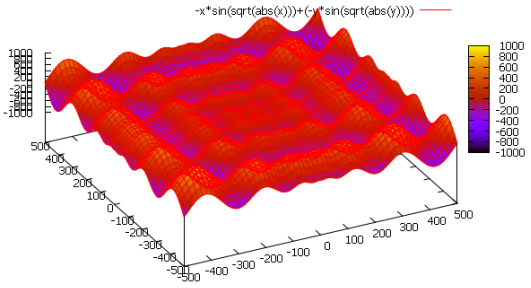
Obrázek 15: Rastrigin: Rozvržení jedinců při poslední iteraci. ABC(vlevo), BA(vpravo)

obrázku 15) vypadají velice podobně, při větším počtu testů se ukázalo, že minimum pro tuto funkci lépe vyhledává algoritmus BA. U algoritmu ABC několikrát nastala situace, při které byly nalezeny pouze lokální extrémy. Zatímco u algoritmu BA se většina včel z elitní skupiny dostala do blízkosti globálního minima. Pro vyhledávání minima této funkce je potřeba správně nastavit velikost prohledávaného okolí. Okolí nesmí být moc veliké, aby byl algoritmus přesný, a zároveň musí být dost veliké, aby byla možnost skočit do okolních extrémů.

5.2.3 Schwefelova funkce

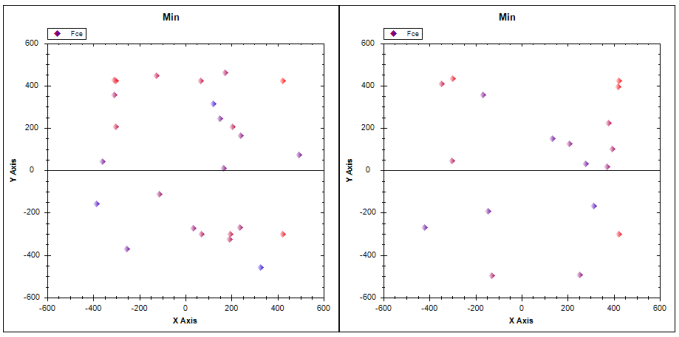
$$\sum_{i=1}^{D} -x_i - \sin\sqrt{|x_i|} \tag{10}$$

Další z testovaných funkcí je Schwefelova(vzorec 10). Tato funkce je zobrazena na ob-



Obrázek 16: Zobrazení Schwefelovy funkce

rázku 16. Globální minimum pro D=2 je přibližně v bodě $(x_1, x_2) = (421, 421)$ a hodnota tohoto minima je $y = -837,966$. Pro větší dimenze je globální minimum přibližně rovno $y = -418,983 \times D$ V této funkci nebyl problém naleznout globální minimum. Oba al-



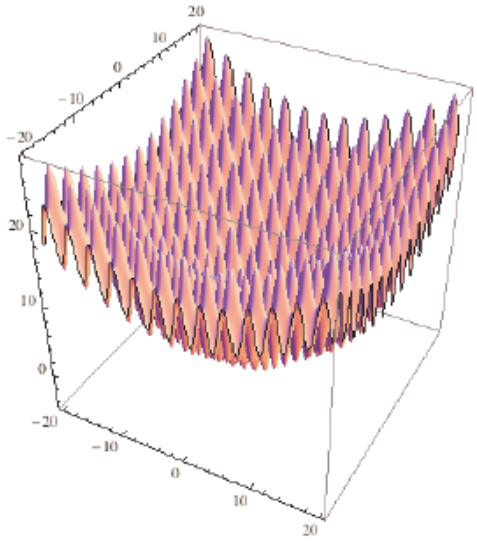
Obrázek 17: Schwefel: Rozvržení jedinců při poslední iteraci. ABC(vlevo), BA(vpravo)

goritmy globální minimum našly bez problémů. Algoritmus BA většinou nacházel minimum dříve, než algoritmus ABC. Algoritmus ABC se na počátku zdržoval v lokálních extrémech, kde promarnil velký počet iterací na nalezení tohoto lokálního extrému. Oba algoritmy jsou schopny většinou nalézt minimum do proběhnutí 100 iterací. Rychlost nalezení minima je závislá na generátoru náhodných pozic. Pokud náhodou generátor nepošle včelu do okolí minima, algoritmus dané minimum nenajde.

5.2.4 Ackleyho funkce 1

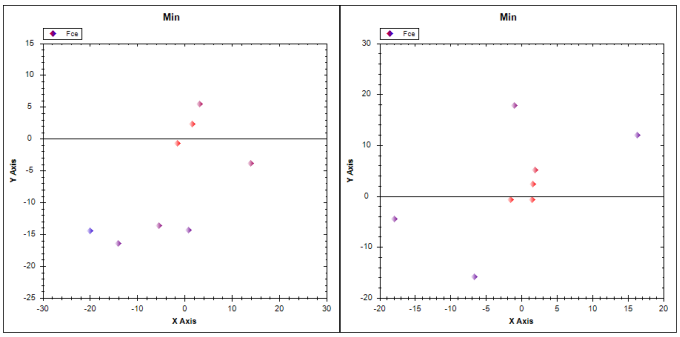
$$\sum_{i=1}^{D-1}(\frac{1}{e^5}\sqrt{(x_i^2+x_{i+1}^2)}+3(\cos 2x_i+\sin 2x_{i+1})) \tag{11}$$

Globální minimum Ackleyho první funkce(vzorec 11) se nachází pro dimenzi D=2 ve



Obrázek 18: Zobrazení první Ackleyho funkce

dvou bodech. První bod je $(x_1, x_2) = (-1,50236; -0,754865))$ a druhý bod je $(x_1, x_2) = (1,50236, -0,754865))$. Toto minimum má přibližnou hodnotu $y = -4,5901$. Pro dimenzi 3 má minimum hodnotu $y = -7,54276$ a pro vyšší dimenzi je minimum popsáno vzorcem $y = -7,54276 - 2,91867 \times (n-3)$



Obrázek 19: Ackley 1: Rozvržení jedinců při poslední iteraci. ABC(vlevo), BA(vpravo)

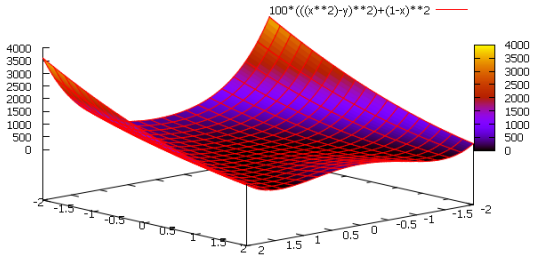
Na obrázku 19 jdou vidět poslední iterace obou algoritmů. Algoritmus ABC(obrázek 19(vlevo)) ve většině pokusů našel pouze jedno minimum. Někdy nastala situace, že

minimum vůbec nenašel. Zatímco BA(obrázek 19(vpravo)) ve většině případu našel obě minima. U ABC jsem se snažil úpravou parametrů dosáhnout podobných výsledků jaku u BA. Po zvýšení iterací, počtu průzkumníků, velikosti lokálního prohledávání a počtu vyčkávacích včel, už i ABC často našlo obě minima.

5.2.5 Druhá de Jongova funkce

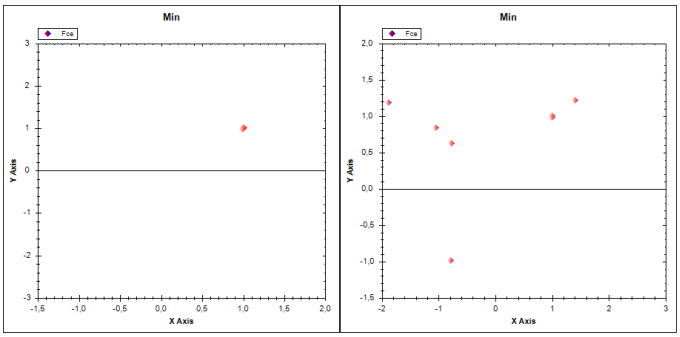
$$\sum_{i=1}^{D-1} 100(x_i^2 - x_{i+1}^2)^2 + (1 - x_i)^2 \tag{12}$$

Druhá de Jongova funkce je popsána vzorcem 12 a její tvar pro dimenzi 2 je zobrazen



Obrázek 20: Zobrazení druhé de Jongovy funkce

na obrázku 20. Globální minimum této funkce je v bodě $(x_1, x_2) = (1, 1))$ a jeho hodnota je $y = 0$. Pro větší rozměry je minimum v bodě $(x_1, x_2, ..., x_D) = (1, 1, ..., 1)$, kde D je dimenze. Hodnota tohoto minima je taktéž $y = 0$. Oba algoritmy se s touto funkci úspěšně



Obrázek 21: de Jong 2: Rozvržení jedinců při poslední iteraci. ABC(vlevo), BA(vpravo)

poradili. Algoritmus BA se k cíli dostal velice rychle. Zatímco algoritmus ABC dostal do minima všechny včely. Na obrázku 21 lze vidět rozdílnost algoritmů. U algoritmu ABC se i nejhorší včela snaží zlepšovat své řešení. U algoritmu BA horší včely své řešení zahodí.

Počet iterací	Dimenze	Průzkumníků	Kolonie	Fitness	Čas
400	50	10	210	0	310ms
400	100	20	310	0	920ms
400	200	30	600	0,053	2,4 s
1000	300	30	2000	0,078	21 s
1000	400	10	1500	0,098	19 s
1000	500	20	2020	0,178	31 s

Tabulka 12: Výsledky testů binárních včel

Funkce	Iterace	Průzkumníků	Velikost okolí	Dimenze	Kolonie
Griewangova	300	10	2	2	140
Rastriginova	300	10	0,3	2	140
Schwefelova	300	30	5	2	330
Ackleyho 1	300	10	0,3	2	140
de Jong 2	300	10	0,3	2	140

Tabulka 13: Nastavení parametrů při testování funkcí

5.3 Testování Binárních včel

Základní verzi binárních včel bez komprese jsem testoval pomocí hledání určitého vektoru. Na začátku algoritmu jsem si vytvořil náhodný vektor, který byl dané dimenze. V průběhu algoritmu jsem u včel používal podobnost hledaného vektoru s vektorem u včely jako fitness hodnotu dané včely. Tuto fitness hodnotu jsem počítal pomocí Jaccardova vzorce popsaného rovnicí 5. Tento vzorec určuje podobnost dvou vektorů. Výsledkem vzorce je číslo mezi 0 a 1. Čím více se blížíme číslu 0, tím jsou vektory podobnější. Číslo 0 znamená, že vektory jsou stejné. V tabulce 12 jsou vypsány výsledky jednotlivých testů.

5.4 Zhodnocení testů

V tabulce 13 Jsou zobrazeny hodnoty parametrů nastavené při testování u obou algoritmů. Oba algoritmy se u všech funkcích dostaly k hledanému výsledku. U některých funkcí bylo potřeba, k nalezení správného výsledku, správně nastavit parametry. Při většině pokusů vycházelo kvalitnější řešení u algoritmu BA.

Dále jsem testoval binární včely. V tabulce 12 lze vidět výsledné hodnoty při různých nastaveních. V menších dimenzích nebyl problém naleznout hledaný vektor velice rychle. U větších dimenzí se se včely vektoru přibližovali, ale nenalezly jej. Nalezení by bylo možné přidáním iterací, vyčkávacích včel a počtu průzkumníků.

6 Experiment

V této kapitole se pokusím aplikovat včelí algoritmus na kompresní algoritmus

6.1 Propojení algoritmů

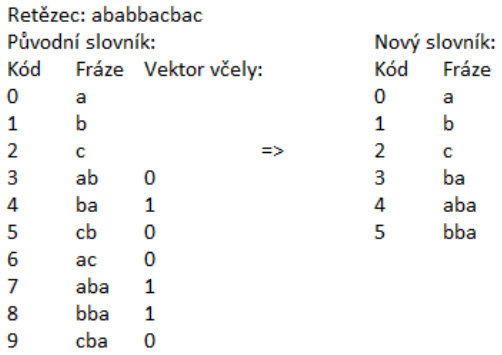
Největší nedostatek u kompresního algoritmu, popsaného v kapitole 3.6, je velikost slovníku. Proto se tedy pokouším tento slovník optimalizovat pomocí binárních včel popsaných v kapitole 4.5.Jako hodnotu fitness jsem u binárních včel používal vzdálenost vektoru od určitého vektoru. Při kompresi nastavuji hodnotu fitness na velikost komprimovaného souboru s daným slovníkem. Tvorbu nového slovníku jsem zachytil na obrázku 22. Základní abecedu nemůžeme omezit, proto je dimenze vektoru u včel rovna počtu frází s délkou větší než 1. Do nového slovníku tedy přepíšeme všechny fráze délky 1. Následně do nového slovníku přidáme pouze fráze u kterých je hodnota včelího vektoru 1.

Jakmile je nový slovník vytvořen je potřeba vypočítat fitness. Tedy je zapotřebí nasimulovat zabalení souboru a zapsat jeho velikost. Tuto simulaci provádím metodou ve které vracím součet bitů potřebných pro uložení slovníku i dat. Mimo vypočítání hodnoty fitness pracuje algoritmus stejně jako při hledání vektoru.

6.2 Testování algoritmu

Pro testování jsem využil několik malých testovacích souborů.
Využité soubory a jejich popis:

- Soubor *alice29.txt* - Tento soubor má velikost 149KB a obsahem je první kapitola knihy Alenka v říši divů.
- Soubor *alphabet.txt* - Tento soubor má velikost 98KB a obsahuje řetězec, ve kterém se opakuje abeceda od a až po z.
- Soubor *asyoulik.txt* - Soubor o velikosti 123KB. Obsahem jsou anglicky psané rozhovory ve tvaru: jméno mezera text.
- Soubor *lcet10.txt* - Soubor s velikostí 417KB, který obsahuje anglicky psaný seminář o elektronických textech
- Soubor *random.txt* - Soubor o velikosti 98KB. Obsahem je řetězec s náhodně vygenerovanými znaky.



Obrázek 22: Tvorba nového slovníku

Velikost sl.	Délka fráze	Iterací	Průzkumníků	Kolonie	Velikost	Čas$[min]$
100	4	50	10	110	102KB	3-4
200	4	50	10	110	90KB	4-5
300	4	50	10	110	92KB	5-6
300	4	100	20	220	90KB	20
300	10	100	10	200	89KB	16
400	10	100	10	200	83KB	18
500	10	100	15	350	77KB	34

Tabulka 14: Výsledky při kompresi souboru asyoulik.txt

6.2.1 Testování na souboru asyoulik.txt

Soubor o velikosti 123KB. Pomocí metody LZW se tento text zkomprimoval na 54KB. Výsledky komprese mým algoritmem jsou zobrazeny v tabulce 14. Nejlepší komprese pomocí testovaného algoritmu byla dosažena při velikosti slovníku 500. Předpokládám, že by včely mohly nalézt i lepší řešení. Ovšem pro lepší řešení bychom potřebovali vyšší počet iterací a větší kolonii. S těmito většími čísly by ovšem stoupl i čas, který už je i nyní velký.

6.2.2 Testování na souboru alphabet.txt

Tento soubor má velikost 98KB. LZW metoda byla schopná soubor zkomprimovat na 5KB. Pro tento soubor nevyšla má komprese o moc hůře. Výsledky jsou zapsány v tabulce 15. Pokud nebudeme brát v potaz poslední řádek této tabulky, jsou i časy komprese celkem uspokojivé. Například když si vezmeme předposlední řádek tabulky. Tento pokus trval 2-3 min. Jelikož je zde 100 iterací a 500 vyčkávacích včel, tak celkový počet balení tohoto souboru je přibližně 50000. Poslední pokus o kompresi sice trval 29 minut, ale opět zvýšil kompresní poměr. I zde odhaduji, že by bylo možné na úkor času dosáhnout lepší komprese.

Velikost sl.	Délka fráze	Iterací	Průzkumníků	Kolonie	Velikost	Čas$[min]$
50	10	50	10	110	39KB	0-1
100	10	50	10	110	23KB	0-1
200	10	50	10	110	13KB	0-1
300	10	100	10	110	12KB	0-1
300	10	100	30	510	12KB	2-3
400	10	100	30	510	11KB	3-4
400	20	100	10	510	7KB	2-3
500	54	500	50	1200	6KB	29

Tabulka 15: Výsledky při kompresi souboru alphabet.txt

Velikost sl.	Délka fráze	Iterací	Průzkumníků	Kolonie	Velikost	Čas$[min]$
66	2	100	10	110	85KB	2-3
70	3	100	10	210	86KB	4-5
100	5	100	10	210	85KB	7-8
127	3	100	10	210	84KB	10-11
200	5	100	10	210	94KB	15-16
300	4	100	20	350	105KB	40-41

Tabulka 16: Výsledky při kompresi souboru random.txt

6.2.3 Testování na souboru random.txt

Soubor o velikosti 98KB. Pomocí LZW jdou tyto data zkomprimovat na 91KB. U tohoto textového souboru zato lépe vyšel můj algoritmus.V tabulce 16 lze vidět, že testovaný algoritmus dokázal porazit LZW i při menších slovnících. Při velikosti slovníku 200 a 300 se komprimovaná velikost začala zvyšovat. Toto zvýšení přisuzuji tomu, že včely nenašly optimální slovník. Pokud by neexistoval lepší slovník, než slovník z předešlého řádku, měly včely najít tento předešlý slovník. Ovšem pro hledání v takové dimenzi bychom museli upravit parametry včel a to by se zvýšila časová náročnost.

6.2.4 Testování na souboru lcet10.txt

Soubor s velikostí 417KB. Metoda LZW tento soubor dokázala zkomprimovat na velikost 160KB. Testovaný algoritmus tyto data zkomprimoval na 309KB. Komprese tohoto souboru byla časově velmi náročná. Jak můžeme vidět v tabulce 17, i při menších hodnotách parametrů trvala komprese 10min. Algoritmus by byl schopen zkomprimovat text daleko líp, ale časová náročnost by byla velice vysoká.

6.2.5 Testování na souboru alice29.txt

Tento soubor má velikost 149KB. LZW dokáže tento text zkomprimovat na 61KB. Mým algoritmem jsem zvládl 93KB, ovšem nastavení parametrů nebylo optimální. Test zobra-

Velikost sl.	Délka fráze	Iterací	Průzkumníků	Kolonie	Velikost	Čas$[min]$
100	3	50	10	110	338KB	10min
200	4	100	10	110	310KB	29min
300	5	100	20	350	309KB	37min

Tabulka 17: Výsledky při kompresi souboru lcet10.txt

Velikost sl.	Délka fráze	Iterací	Průzkumníků	Kolonie	Velikost	Čas$[min]$
100	3	50	10	110	122KB	3min
200	3	100	10	110	100KB	8min
300	4	100	15	220	100KB	19min
300	5	100	10	310	99KB	24min
400	5	200	10	310	93KB	59min

Tabulka 18: Výsledky při kompresi souboru alice29.txt

zený na posledním řádku tabulky 18 trval 59 minut. Jako optimální nastavení počítám přibližně velikost slovníku 500 - 1000, průzkumníků 50 - 100, kolonie 5000 - 7000 a iterací 500 - 1000. S těmito parametry by kompresní pomět měl být lepší(v blízkosti nejlepšího).

7 Závěr

Záměrem diplomové práce bylo zkombinovat kompresi dat se včelími algoritmy. Na začátku práce jsou sepsány základní metody optimalizace pomocí včelích kolonií a komprese dat. Práce se rozděluje do dvou částí. První část se zabývá implementací metod pro optimalizaci pomocí včelích kolonií a následnému testování těchto metod na klasických optimalizačních funkcích. Druhá část se zabývá propojením včelího algoritmu s kompresním algoritmem.

Chování algoritmů u klasických optimalizačních metod jsem otestoval na různých vícerozměrných funkcích. Na těchto funkcích jsem hledal globální minima. Algoritmy nacházely tyto minima většinou bez větší problémů. Nejlepších výsledků dosahoval algoritmus BA(Bee Algorithm), který nacházel minima dříve a přesněji.

Následná část obsahuje přepracování včelího algoritmu do verze, která pracuje s binárními čísly. Přepracování nebylo složité. Největší problém nastal při prohledávání okolí, kde bylo potřeba definovat, jak bude vypadat okolí u aktuálního vektoru. Pro toto prohledávání jsem nakonec použil vzorec, který využívá náhodné včely a operace XOR. Jakmile byl vytvořen binární včelí algoritmus propojil jsem ho s kompresí a mohl jsem začít testovat.

Chování kompletního algoritmu jsem testoval na skupině testovacích souborů různých druhů. Na některých algoritmech se vytvořená metoda přibližovala kompresnímu poměru metody LZW. Testovaný algoritmus porazil metodu LZW u souboru „random.txt“. Tento soubor obsahoval náhodné znaky. Důvod lepší komprese přikládám tomu, že si algoritmus vytvořil slovník o velikosti 127 řádků, kde bylo přibližně 64 řádků s frázemi délky 1 a zbylé řádky obsahovaly dvojznaky, které měly nejčastější výskyt. Jelikož byla délka slovníku 127, bylo potřeba pro zapsání každého indexu 7 bitů, proto je komprese účinná. U větších souborů měl algoritmus menší kompresní poměr. Tento poměr by se dal zvětšit úpravou parametrů včel, ovšem na úkor času.

Biologicky inspirované algoritmy jsou v této době aktuální téma. I když táto práce nedospěla k nějakým překvapivým výsledkům, určitě je potřeba dále zkoumat, kde v kompresi by se dali biologicky inspirované algoritmy využít. Například by se v této práci dala využít Soma, nebo Diferenciální evoluce. Tyto metody by mohly dosahovat lepších výsledků, než mých.

8 Reference

[1] *ZedGraph* [online]. 2008-12-12 [cit. 2013-04-7] Dostupné z: <http://sourceforge.net/projects/zedgraph/>.

[2] VEČERKA, Arnošt. *KOMPRESE DAT* [online]. 2008 [cit. 2013-04-11]. Dostupné z: http://phoenix.inf.upol.cz/esf/ucebni/komprese.pdf

[3] ZELINKA, I., Z. OPLATKOVÁ, P OŠMERA, M. ŠEDA a F. VČELAŘ.e, *Evoluční výpočetní techniky: principy a aplikace.* Praha: BEN - technická literatura, 2008. ISBN 80-7300-218-3.

[4] *Mersenne Twister: A random number generator (since 1997/10)* [online]. [1998-2007] [cit. 2013-04-20]. Dostupné z: http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html

[5] KASHAN, Mina Husseinzadeh, Nasim NAHAVANDI a Ali Husseinzadeh KASHAN. DisABC: A new artificial bee colony algorithm for binary optimization. *Applied Soft Computing* [online]. roč. 12, č. 1, s. 342-352 [cit. 2013-04-22]. ISSN 15684946. DOI: 10.1016/j.asoc.2011.08.038. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S1568494611003218

[6] NAGEL, Christian, et al. *C# 2008 : Programujme profesionálně* Vyd. 1. Brno : Computer Press, 2009. 1126 s. ISBN 978-80-251-2401-7.

A Příloha

Součásti originálu diplomové práce je CDROM obsahující tuto práci a veškeré zdrojové kódy.

A.1 Obsah CD

- V adresáři Program je pouze program.
- V adresáři Zdrojový kód je program se všemi zdrojovými kódy.
- V adresáři Diplomová práce je tato práce v elektronické podobě.